/\*\*

\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Bailey Thompson

\* Maze (1.1)

\* 16 September 2016

\* Info: The user is first introduced to a default grid, in which a file is created using file io. The user has an option of three

\* Info: buttons and two sliders on the bottom. When the clear button is pressed, the board is reset to only walls and paths, when

\* Info: the generate button is pressed, a new board of specified size is created, and when the exit button is pressed, the program

\* Info: exits. The size slider is a value between and including 2 to 30, when the generate button is pressed, the size is thus

\* Info: reflected, if the user hovers over the slider, important information is displayed to the user. The time slider is between

\* Info: and including 0 to 1000 -- the time is in milliseconds; 0 is instant -- the time slider is reflected immediately after it

\* Info: is changed, as with the other slider this one also displays important information if hovered over. The first click on the

\* Info: board is the start position and is in red. The second click is the end position in blue. A green cell will go from start

\* Info: to end and change cell once per turn as specified by the time slider. Once the green cell reached the end, it will display

\* Info: the path taken. When generate is clicked, the progress in percentage is displayed next to the title of the program.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*/

//declaring package

package maze;

//declaring imports

import java.awt.BorderLayout;

import java.awt.Color;

import java.awt.Dimension;

import java.awt.FlowLayout;

import java.awt.Graphics;

import java.awt.Graphics2D;

import java.awt.Point;

import java.awt.Rectangle;

import java.awt.Toolkit;

import java.awt.event.ActionEvent;

import java.awt.event.MouseAdapter;

import java.awt.event.MouseEvent;

import java.io.BufferedOutputStream;

import java.io.BufferedReader;

import java.io.IOException;

import java.io.InputStream;

import java.io.InputStreamReader;

import java.io.OutputStream;

import static java.lang.Integer.parseInt;

import java.nio.file.FileAlreadyExistsException;

import java.nio.file.Files;

import java.nio.file.Path;

import java.nio.file.Paths;

import static java.nio.file.StandardOpenOption.TRUNCATE\_EXISTING;

import static java.nio.file.StandardOpenOption.WRITE;

import java.util.ArrayList;

import java.util.List;

import java.util.logging.Level;

import java.util.logging.Logger;

import javax.swing.JButton;

import javax.swing.JFrame;

import javax.swing.JPanel;

import javax.swing.JSlider;

//declaring class

public class Maze {

//declaring name of path file

Path file = Paths.get("RecursiveMazeSolver.txt");

//declaring String array used for file io

String[] split;

//declaring variables used for colouring in cells for the gui and for recursively solving the maze

int xOffset, yOffset, colourMode, currentX, currentY, endX, endY, startX, startY, guiDisplay, sizeValue, time, positionCounter;

//used for recursively solving the maze

char direction;

//declaring variable for amount of tries has been attemped to generate maze

long tries;

//declaring variable for the percentage completion of the loading

double percentage;

//various gui components

private JFrame frame;

private JSlider sizeSlider, timingSlider;

private JPanel panel;

private JButton btnClear, btnGenerate, btnExit;

//declaring variable for the status of each cell in the maze

int[][] mazeArray, positionArray;

//declaring variable used for generating the maze

boolean[][] visitedArray;

//declaring variables for setting the colour of a cell

private List<Rectangle> cells;

private Point selectedCell;

//used to only reference method MakePath once from method StartSolver

boolean firstTime;

//variable used for file io

String saveFile;

//declaring main method

public static void main(String[] args) {

//sending to Maze method

Maze Maze = new Maze();

Maze.Maze();

}

//declaring private void method for referencing various methods

private void Maze() {

//checking the monitor dimensions

Dimension screenSize = Toolkit.getDefaultToolkit().getScreenSize();

//setting the gui size

if (screenSize.getWidth() < screenSize.getHeight()) {

guiDisplay = (int) (screenSize.getWidth() \* 0.8);

} else {

guiDisplay = (int) (screenSize.getHeight() \* 0.8);

}

//sending to method Load

Load();

//setting the size of the maze array

mazeArray = new int[sizeValue][sizeValue];

//setting size of position array

positionArray = new int[sizeValue][sizeValue];

//setting the size of the visited array

visitedArray = new boolean[sizeValue][sizeValue];

//variables for setting of cell

for (int vertical = 0; vertical < sizeValue; vertical++) {

for (int horizontal = 0; horizontal < sizeValue; horizontal++) {

mazeArray[vertical][horizontal] = parseInt(split[vertical \* sizeValue + horizontal + 2], 10);

}

}

//sending to method PrepareGUI

PrepareGUI();

}

//declaring private void method used for creating the gui

private void PrepareGUI() {

//setting the frame title

frame = new JFrame("Maze");

//making it so when the x button is pressed the program exits

frame.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

//making the frame non-resizable

frame.setResizable(false);

//making the GUI more user-friendly

frame.setLayout(new BorderLayout());

frame.add(new GridPane());

frame.pack();

//centering the frame

frame.setLocationRelativeTo(null);

//setting the row of buttons

panel = new JPanel();

//setting buttons and what is displayed on them

btnClear = new JButton("Clear");

btnGenerate = new JButton("Generate");

btnExit = new JButton("Exit");

//creating a new horizontal slider from 2 to 30 with a starting position of the number read with file io

sizeSlider = new JSlider(JSlider.HORIZONTAL, 2, 30, sizeValue);

//setting the slider to print numbers under it

sizeSlider.setPaintLabels(true);

//setting how often numbers are printed

sizeSlider.setMajorTickSpacing(4);

//setting dimensions of sliderSize variable

sizeSlider.setPreferredSize(new Dimension(150, 40));

//setting what is displayed when user hovers over slider with mouse

sizeSlider.setToolTipText("Size Of The Grid");

//creating a new horizontal slider from 0 to 1000 with a starting position of the number read with file io

timingSlider = new JSlider(JSlider.HORIZONTAL, 0, 1000, time);

//setting the slider to print numbers under it

timingSlider.setPaintLabels(true);

//setting how often numbers are printed

timingSlider.setMajorTickSpacing(250);

//setting dimensions of timingSlider variable

timingSlider.setPreferredSize(new Dimension(150, 40));

//setting what is displayed when user hovers over slider with mouse

timingSlider.setToolTipText("Miliseconds Between Turns");

//setting the layout of both rows of buttons

panel.setLayout(new FlowLayout(FlowLayout.CENTER));

//setting upper row of buttons to variable panel

panel.add(sizeSlider);

panel.add(btnClear);

panel.add(btnGenerate);

panel.add(btnExit);

panel.add(timingSlider);

//setting various parts of the frame

frame.add(panel, BorderLayout.SOUTH);

//setting the frame to visible

frame.setVisible(true);

//setting what happens when user clicks on clear button

btnClear.addActionListener((ActionEvent e) -> {

//setting variables back to default

for (int vertical = 0; vertical < sizeValue; vertical++) {

for (int horizontal = 0; horizontal < sizeValue; horizontal++) {

if (mazeArray[vertical][horizontal] != 1) {

mazeArray[vertical][horizontal] = 0;

positionArray[vertical][horizontal] = 0;

}

}

}

//setting various variables to default settings

firstTime = false;

colourMode = positionCounter = 0;

});

//setting what happens when user clicks on generate button

btnGenerate.addActionListener((ActionEvent e) -> {

//setting various variables to default settings

firstTime = false;

percentage = colourMode = positionCounter = 0;

tries = 0;

//setting value of sizeValue

sizeValue = sizeSlider.getValue();

//setting the size of the maze array

mazeArray = new int[sizeValue][sizeValue];

//setting size of position array

positionArray = new int[sizeValue][sizeValue];

//setting the size of the visited array

visitedArray = new boolean[sizeValue][sizeValue];

//setting the size of the array list used in the graphical output

cells = new ArrayList<>(sizeValue \* sizeValue);

//sending to method Randomize

Randomize();

//sending to method Save

Save();

});

//setting what happens when user clicks on exit button

btnExit.addActionListener((ActionEvent e) -> {

//program exits

System.exit(0);

});

}

//declaring private void method used for generating maze

private void Randomize() {

//creating variables used for determining amount of white and black cells

int white, black;

//loop executed then executed again if too many black tiles

do {

//sending to method InitializeRandomize

InitializeRandomize();

//setting variables to zero

white = black = 0;

//using 2d array to check every single cells

for (int vertical = 0; vertical < sizeValue; vertical++) {

for (int horizontal = 0; horizontal < sizeValue; horizontal++) {

//adding one to white if there is a white cell present

if (mazeArray[vertical][horizontal] == 0) {

white += 1;

//adding one to black if there is a black cell present

} else if (mazeArray[vertical][horizontal] == 1) {

black += 1;

}

}

}

} while (black / white > 0.5);

//displaying the title of the program

frame.setTitle("Maze");

}

//declaring declaring private void method used for setting start cell

private void InitializeRandomize() {

//setting every cell to wall and unvisited

for (int vertical = 0; vertical < sizeValue; vertical++) {

for (int horizontal = 0; horizontal < sizeValue; horizontal++) {

mazeArray[vertical][horizontal] = 1;

visitedArray[vertical][horizontal] = false;

}

}

//generating two random numbers

int randomOne = (int) (Math.random() \* sizeValue);

int randomTwo = (int) (Math.random() \* sizeValue);

//setting the start tile to path and visited

mazeArray[randomOne][randomTwo] = 0;

visitedArray[randomOne][randomTwo] = true;

//creating various temporary tiles around the seed tile

if (randomOne > 0) {

mazeArray[randomOne - 1][randomTwo] = 10;

}

if (randomOne < sizeValue - 1) {

mazeArray[randomOne + 1][randomTwo] = 10;

}

if (randomTwo > 0) {

mazeArray[randomOne][randomTwo - 1] = 10;

}

if (randomTwo < sizeValue - 1) {

mazeArray[randomOne][randomTwo + 1] = 10;

}

//sending to method RandomGenerator

RandomGenerator();

}

//declaring private void method used for creating path cells from seed tile

private void RandomGenerator() {

//creating and setting various variables

boolean skip = false, allDone = true;

int wallCells = 0;

//2d array used for setting the amount of wall cells to a variable

for (int vertical = 0; vertical < sizeValue; vertical++) {

for (int horizontal = 0; horizontal < sizeValue; horizontal++) {

if (mazeArray[vertical][horizontal] == 10) {

wallCells += 1;

}

}

}

//2d array used for creating path cells

for (int vertical = 0; vertical < sizeValue; vertical++) {

for (int horizontal = 0; horizontal < sizeValue; horizontal++) {

//random generation used for detemining if cell should be picked

int randomPass = (int) (Math.random() \* (wallCells + 1));

//only uses cell if random generation picks cell and if cells this turn has not already been picked and if the cell is actually a temp cell

if (skip == false && mazeArray[vertical][horizontal] == 10 && randomPass == 0) {

//declaring and setting variable to zero

int neighbours = 0;

//checking all four sides of cell and reporting amount fo neighbours

if (vertical > 0) {

if (mazeArray[vertical - 1][horizontal] == 0) {

neighbours += 1;

}

}

if (vertical < sizeValue - 1) {

if (mazeArray[vertical + 1][horizontal] == 0) {

neighbours += 1;

}

}

if (horizontal > 0) {

if (mazeArray[vertical][horizontal - 1] == 0) {

neighbours += 1;

}

}

if (horizontal < sizeValue - 1) {

if (mazeArray[vertical][horizontal + 1] == 0) {

neighbours += 1;

}

}

//setting if cells is full or if it empty depending on amount of neighbours

if (neighbours == 1) {

mazeArray[vertical][horizontal] = 0;

} else {

mazeArray[vertical][horizontal] = 1;

}

//setting the temp cells around the cell

if (vertical > 0) {

if (visitedArray[vertical - 1][horizontal] == false) {

mazeArray[vertical - 1][horizontal] = 10;

}

}

if (vertical < sizeValue - 1) {

if (visitedArray[vertical + 1][horizontal] == false) {

mazeArray[vertical + 1][horizontal] = 10;

}

}

if (horizontal > 0) {

if (visitedArray[vertical][horizontal - 1] == false) {

mazeArray[vertical][horizontal - 1] = 10;

}

}

if (horizontal < sizeValue - 1) {

if (visitedArray[vertical][horizontal + 1] == false) {

mazeArray[vertical][horizontal + 1] = 10;

}

}

//setting skip to true so that this round only one cell is created

skip = true;

//setting the cell to visited

visitedArray[vertical][horizontal] = true;

}

}

}

//determining if the maze is completed

for (int vertical = 0; vertical < sizeValue; vertical++) {

for (int horizontal = 0; horizontal < sizeValue; horizontal++) {

if (visitedArray[vertical][horizontal] == false) {

allDone = false;

}

}

}

//setting the percentage to the user using an algorithm

//from 0% to 80% it is normal speed

//from 80% to 90% it is half speed

//from 90% to 95% it is a quarter speed

//from 95% to 99% it is one eight speed

//it never goes above 99%

if (allDone == false) {

tries += 1;

if (percentage != 99) {

percentage = (100 \* tries) / (Math.pow(48, (2 \* sizeValue - 10) \* 0.05 + 1));

if (percentage > 80) {

percentage = (percentage - 80) / 2 + 80;

if (percentage > 90) {

percentage = (percentage - 90) / 4 + 90;

if (percentage > 95) {

percentage = (percentage - 95) / 8 + 95;

if (percentage > 99) {

percentage = 99;

}

}

}

}

//displaying for the user to wait and showing percentage

frame.setTitle("Maze (" + (int) percentage + "% Done Loading)");

}

//recursively sending to method RandomGenerator

RandomGenerator();

}

}

//declaring private void method used to solve the maze using the algorithm I made

private void StartSolver() {

//only do such if the maze is not solved

if (Math.abs(endX - currentX) + Math.abs(endY - currentY) != 1) {

//case for if in the middle of the board and not corners or sides

if (currentX > 0 && currentX < sizeValue - 1 && currentY > 0 && currentY < sizeValue - 1) {

switch (direction) {

case 'r':

//know they were going right

if (mazeArray[currentY + 1][currentX] != 1) { //going down

currentY += 1;

direction = 'd';

} else if (mazeArray[currentY][currentX + 1] != 1) { //going right

currentX += 1;

direction = 'r';

} else if (mazeArray[currentY - 1][currentX] != 1) { //going up

currentY -= 1;

direction = 'u';

} else if (mazeArray[currentY][currentX - 1] != 1) { //going left

currentX -= 1;

direction = 'l';

}

break;

case 'l':

//know they were going left

if (mazeArray[currentY - 1][currentX] != 1) { //going up

currentY -= 1;

direction = 'u';

} else if (mazeArray[currentY][currentX - 1] != 1) { //going left

currentX -= 1;

direction = 'l';

} else if (mazeArray[currentY + 1][currentX] != 1) { //going down

currentY += 1;

direction = 'd';

} else if (mazeArray[currentY][currentX + 1] != 1) { //going right

currentX += 1;

direction = 'r';

}

break;

case 'd':

//know they were going down

if (mazeArray[currentY][currentX - 1] != 1) { //going left

currentX -= 1;

direction = 'l';

} else if (mazeArray[currentY + 1][currentX] != 1) { //going down

currentY += 1;

direction = 'd';

} else if (mazeArray[currentY][currentX + 1] != 1) { //going right

currentX += 1;

direction = 'r';

} else if (mazeArray[currentY - 1][currentX] != 1) { //going up

currentY -= 1;

direction = 'u';

}

break;

case 'u':

//know they were going up

if (mazeArray[currentY][currentX + 1] != 1) { //going right

currentX += 1;

direction = 'r';

} else if (mazeArray[currentY - 1][currentX] != 1) { //going up

currentY -= 1;

direction = 'u';

} else if (mazeArray[currentY][currentX - 1] != 1) { //going left

currentX -= 1;

direction = 'l';

} else if (mazeArray[currentY + 1][currentX] != 1) { //going down

currentY += 1;

direction = 'd';

}

break;

}

//case for if on side of board

} else if (currentX == 0 && currentY > 0 && currentY < sizeValue - 1) {

switch (direction) {

case 'r':

//know they were going right

if (mazeArray[currentY + 1][currentX] != 1) { //going down

currentY += 1;

direction = 'd';

} else if (mazeArray[currentY][currentX + 1] != 1) { //going right

currentX += 1;

direction = 'r';

} else if (mazeArray[currentY - 1][currentX] != 1) { //going up

currentY -= 1;

direction = 'u';

}

break;

case 'l':

//know they were going left

if (mazeArray[currentY - 1][currentX] != 1) { //going up

currentY -= 1;

direction = 'u';

} else if (mazeArray[currentY + 1][currentX] != 1) { //going down

currentY += 1;

direction = 'd';

} else if (mazeArray[currentY][currentX + 1] != 1) { //going right

currentX += 1;

direction = 'r';

}

break;

case 'd':

//know they were going down

if (mazeArray[currentY + 1][currentX] != 1) { //going down

currentY += 1;

direction = 'd';

} else if (mazeArray[currentY][currentX + 1] != 1) { //going right

currentX += 1;

direction = 'r';

} else if (mazeArray[currentY - 1][currentX] != 1) { //going up

currentY -= 1;

direction = 'u';

}

break;

case 'u':

//know they were going up

if (mazeArray[currentY][currentX + 1] != 1) { //going right

currentX += 1;

direction = 'r';

} else if (mazeArray[currentY - 1][currentX] != 1) { //going up

currentY -= 1;

direction = 'u';

} else if (mazeArray[currentY + 1][currentX] != 1) { //going down

currentY += 1;

direction = 'd';

}

break;

}

//case for if on side of board

} else if (currentX == sizeValue - 1 && currentY > 0 && currentY < sizeValue - 1) {

switch (direction) {

case 'r':

//know they were going right

if (mazeArray[currentY + 1][currentX] != 1) { //going down

currentY += 1;

direction = 'd';

} else if (mazeArray[currentY - 1][currentX] != 1) { //going up

currentY -= 1;

direction = 'u';

} else if (mazeArray[currentY][currentX - 1] != 1) { //going left

currentX -= 1;

direction = 'l';

}

break;

case 'l':

//know they were going left

if (mazeArray[currentY - 1][currentX] != 1) { //going up

currentY -= 1;

direction = 'u';

} else if (mazeArray[currentY][currentX - 1] != 1) { //going left

currentX -= 1;

direction = 'l';

} else if (mazeArray[currentY + 1][currentX] != 1) { //going down

currentY += 1;

direction = 'd';

}

break;

case 'd':

//know they were going down

if (mazeArray[currentY][currentX - 1] != 1) { //going left

currentX -= 1;

direction = 'l';

} else if (mazeArray[currentY + 1][currentX] != 1) { //going down

currentY += 1;

direction = 'd';

} else if (mazeArray[currentY - 1][currentX] != 1) { //going up

currentY -= 1;

direction = 'u';

}

break;

case 'u':

//know they were going up

if (mazeArray[currentY - 1][currentX] != 1) { //going up

currentY -= 1;

direction = 'u';

} else if (mazeArray[currentY][currentX - 1] != 1) { //going left

currentX -= 1;

direction = 'l';

} else if (mazeArray[currentY + 1][currentX] != 1) { //going down

currentY += 1;

direction = 'd';

}

break;

}

//case for if on side of board

} else if (currentY == 0 && currentX > 0 && currentX < sizeValue - 1) {

switch (direction) {

case 'r':

//know they were going right

if (mazeArray[currentY + 1][currentX] != 1) { //going down

currentY += 1;

direction = 'd';

} else if (mazeArray[currentY][currentX + 1] != 1) { //going right

currentX += 1;

direction = 'r';

} else if (mazeArray[currentY][currentX - 1] != 1) { //going left

currentX -= 1;

direction = 'l';

}

break;

case 'l':

//know they were going left

if (mazeArray[currentY][currentX - 1] != 1) { //going left

currentX -= 1;

direction = 'l';

} else if (mazeArray[currentY + 1][currentX] != 1) { //going down

currentY += 1;

direction = 'd';

} else if (mazeArray[currentY][currentX + 1] != 1) { //going right

currentX += 1;

direction = 'r';

}

break;

case 'd':

//know they were going down

if (mazeArray[currentY][currentX - 1] != 1) { //going left

currentX -= 1;

direction = 'l';

} else if (mazeArray[currentY + 1][currentX] != 1) { //going down

currentY += 1;

direction = 'd';

} else if (mazeArray[currentY][currentX + 1] != 1) { //going right

currentX += 1;

direction = 'r';

}

break;

case 'u':

//know they were going up

if (mazeArray[currentY][currentX + 1] != 1) { //going right

currentX += 1;

direction = 'r';

} else if (mazeArray[currentY][currentX - 1] != 1) { //going left

currentX -= 1;

direction = 'l';

} else if (mazeArray[currentY + 1][currentX] != 1) { //going down

currentY += 1;

direction = 'd';

}

break;

}

//case for if on side of board

} else if (currentY == sizeValue - 1 && currentX > 0 && currentX < sizeValue - 1) {

switch (direction) {

case 'r':

//know they were going right

if (mazeArray[currentY][currentX + 1] != 1) { //going right

currentX += 1;

direction = 'r';

} else if (mazeArray[currentY - 1][currentX] != 1) { //going up

currentY -= 1;

direction = 'u';

} else if (mazeArray[currentY][currentX - 1] != 1) { //going left

currentX -= 1;

direction = 'l';

}

break;

case 'l':

//know they were going left

if (mazeArray[currentY - 1][currentX] != 1) { //going up

currentY -= 1;

direction = 'u';

} else if (mazeArray[currentY][currentX - 1] != 1) { //going left

currentX -= 1;

direction = 'l';

} else if (mazeArray[currentY][currentX + 1] != 1) { //going right

currentX += 1;

direction = 'r';

}

break;

case 'd':

//know they were going down

if (mazeArray[currentY][currentX - 1] != 1) { //going left

currentX -= 1;

direction = 'l';

} else if (mazeArray[currentY][currentX + 1] != 1) { //going right

currentX += 1;

direction = 'r';

} else if (mazeArray[currentY - 1][currentX] != 1) { //going up

currentY -= 1;

direction = 'u';

}

break;

case 'u':

//know they were going up

if (mazeArray[currentY][currentX + 1] != 1) { //going right

currentX += 1;

direction = 'r';

} else if (mazeArray[currentY - 1][currentX] != 1) { //going up

currentY -= 1;

direction = 'u';

} else if (mazeArray[currentY][currentX - 1] != 1) { //going left

currentX -= 1;

direction = 'l';

}

break;

}

//case for if in corner of board

} else if (currentX == 0 && currentY == 0) {

switch (direction) {

case 'r':

//know they were going right

if (mazeArray[currentY + 1][currentX] != 1) { //going down

currentY += 1;

direction = 'd';

} else if (mazeArray[currentY][currentX + 1] != 1) { //going right

currentX += 1;

direction = 'r';

}

break;

case 'l':

//know they were going left

if (mazeArray[currentY + 1][currentX] != 1) { //going down

currentY += 1;

direction = 'd';

} else if (mazeArray[currentY][currentX + 1] != 1) { //going right

currentX += 1;

direction = 'r';

}

break;

case 'd':

//know they were going down

if (mazeArray[currentY + 1][currentX] != 1) { //going down

currentY += 1;

direction = 'd';

} else if (mazeArray[currentY][currentX + 1] != 1) { //going right

currentX += 1;

direction = 'r';

}

break;

case 'u':

//know they were going up

if (mazeArray[currentY][currentX + 1] != 1) { //going right

currentX += 1;

direction = 'r';

} else if (mazeArray[currentY + 1][currentX] != 1) { //going down

currentY += 1;

direction = 'd';

}

break;

}

//case for if in corner of board

} else if (currentX == 0 && currentY == sizeValue - 1) {

switch (direction) {

case 'r':

//know they were going right

if (mazeArray[currentY][currentX + 1] != 1) { //going right

currentX += 1;

direction = 'r';

} else if (mazeArray[currentY - 1][currentX] != 1) { //going up

currentY -= 1;

direction = 'u';

}

break;

case 'l':

//know they were going left

if (mazeArray[currentY - 1][currentX] != 1) { //going up

currentY -= 1;

direction = 'u';

} else if (mazeArray[currentY][currentX + 1] != 1) { //going right

currentX += 1;

direction = 'r';

}

break;

case 'd':

//know they were going down

if (mazeArray[currentY][currentX + 1] != 1) { //going right

currentX += 1;

direction = 'r';

} else if (mazeArray[currentY - 1][currentX] != 1) { //going up

currentY -= 1;

direction = 'u';

}

break;

case 'u':

//know they were going up

if (mazeArray[currentY][currentX + 1] != 1) { //going right

currentX += 1;

direction = 'r';

} else if (mazeArray[currentY - 1][currentX] != 1) { //going up

currentY -= 1;

direction = 'u';

}

break;

}

//case for if in corner of board

} else if (currentX == sizeValue - 1 && currentY == 0) {

switch (direction) {

case 'r':

//know they were going right

if (mazeArray[currentY + 1][currentX] != 1) { //going down

currentY += 1;

direction = 'd';

} else if (mazeArray[currentY][currentX - 1] != 1) { //going left

currentX -= 1;

direction = 'l';

}

break;

case 'l':

//know they were going left

if (mazeArray[currentY][currentX - 1] != 1) { //going left

currentX -= 1;

direction = 'l';

} else if (mazeArray[currentY + 1][currentX] != 1) { //going down

currentY += 1;

direction = 'd';

}

break;

case 'd':

//know they were going down

if (mazeArray[currentY][currentX - 1] != 1) { //going left

currentX -= 1;

direction = 'l';

} else if (mazeArray[currentY + 1][currentX] != 1) { //going down

currentY += 1;

direction = 'd';

}

break;

case 'u':

//know they were going up

if (mazeArray[currentY][currentX - 1] != 1) { //going left

currentX -= 1;

direction = 'l';

} else if (mazeArray[currentY + 1][currentX] != 1) { //going down

currentY += 1;

direction = 'd';

}

break;

}

//case for if in corner of board

} else if (currentX == sizeValue - 1 && currentY == sizeValue - 1) {

switch (direction) {

case 'r':

//know they were going right

if (mazeArray[currentY - 1][currentX] != 1) { //going up

currentY -= 1;

direction = 'u';

} else if (mazeArray[currentY][currentX - 1] != 1) { //going left

currentX -= 1;

direction = 'l';

}

break;

case 'l':

//know they were going left

if (mazeArray[currentY - 1][currentX] != 1) { //going up

currentY -= 1;

direction = 'u';

} else if (mazeArray[currentY][currentX - 1] != 1) { //going left

currentX -= 1;

direction = 'l';

}

break;

case 'd':

//know they were going down

if (mazeArray[currentY][currentX - 1] != 1) { //going left

currentX -= 1;

direction = 'l';

} else if (mazeArray[currentY - 1][currentX] != 1) { //going up

currentY -= 1;

direction = 'u';

}

break;

case 'u':

//know they were going up

if (mazeArray[currentY - 1][currentX] != 1) { //going up

currentY -= 1;

direction = 'u';

} else if (mazeArray[currentY][currentX - 1] != 1) { //going left

currentX -= 1;

direction = 'l';

}

break;

}

}

//set the current position to 5

mazeArray[currentY][currentX] = 5;

//if the start point is no longer marked, re-mark it

if (mazeArray[startY][startX] != 2) {

mazeArray[startY][startX] = 2;

}

StartSolver();

}

//only execute once

if (firstTime == false) {

//changes the direction so that it can find most direct path

switch (direction) {

case 'u':

direction = 'd';

break;

case 'd':

direction = 'u';

break;

case 'l':

direction = 'r';

break;

case 'r':

direction = 'l';

break;

}

//setting the current points to the end points

currentX = endX;

currentY = endY;

//sending to method MakePath

MakePath();

//making it so this is not executed again

firstTime = true;

}

}

//declaring private void method used for checking the shortest path

private void MakePath() {

//only do such if shortest path is not yet found

if (Math.abs(startX - currentX) + Math.abs(startY - currentY) != 1) {

//situation for when cell is in middle of board and not corner or side

if (currentX > 0 && currentX < sizeValue - 1 && currentY > 0 && currentY < sizeValue - 1) {

switch (direction) {

case 'r':

//know they were going right

if (mazeArray[currentY + 1][currentX] == 5) { //going down

currentY += 1;

direction = 'd';

} else if (mazeArray[currentY][currentX + 1] == 5) { //going right

currentX += 1;

direction = 'r';

} else if (mazeArray[currentY - 1][currentX] == 5) { //going up

currentY -= 1;

direction = 'u';

} else if (mazeArray[currentY][currentX - 1] == 5) { //going left

currentX -= 1;

direction = 'l';

}

break;

case 'l':

//know they were going left

if (mazeArray[currentY - 1][currentX] == 5) { //going up

currentY -= 1;

direction = 'u';

} else if (mazeArray[currentY][currentX - 1] == 5) { //going left

currentX -= 1;

direction = 'l';

} else if (mazeArray[currentY + 1][currentX] == 5) { //going down

currentY += 1;

direction = 'd';

} else if (mazeArray[currentY][currentX + 1] == 5) { //going right

currentX += 1;

direction = 'r';

}

break;

case 'd':

//know they were going down

if (mazeArray[currentY][currentX - 1] == 5) { //going left

currentX -= 1;

direction = 'l';

} else if (mazeArray[currentY + 1][currentX] == 5) { //going down

currentY += 1;

direction = 'd';

} else if (mazeArray[currentY][currentX + 1] == 5) { //going right

currentX += 1;

direction = 'r';

} else if (mazeArray[currentY - 1][currentX] == 5) { //going up

currentY -= 1;

direction = 'u';

}

break;

case 'u':

//know they were going up

if (mazeArray[currentY][currentX + 1] == 5) { //going right

currentX += 1;

direction = 'r';

} else if (mazeArray[currentY - 1][currentX] == 5) { //going up

currentY -= 1;

direction = 'u';

} else if (mazeArray[currentY][currentX - 1] == 5) { //going left

currentX -= 1;

direction = 'l';

} else if (mazeArray[currentY + 1][currentX] == 5) { //going down

currentY += 1;

direction = 'd';

}

break;

}

//situation for when cell is on side of board

} else if (currentX == 0 && currentY > 0 && currentY < sizeValue - 1) {

switch (direction) {

case 'r':

//know they were going right

if (mazeArray[currentY + 1][currentX] == 5) { //going down

currentY += 1;

direction = 'd';

} else if (mazeArray[currentY][currentX + 1] == 5) { //going right

currentX += 1;

direction = 'r';

} else if (mazeArray[currentY - 1][currentX] == 5) { //going up

currentY -= 1;

direction = 'u';

}

break;

case 'l':

//know they were going left

if (mazeArray[currentY - 1][currentX] == 5) { //going up

currentY -= 1;

direction = 'u';

} else if (mazeArray[currentY + 1][currentX] == 5) { //going down

currentY += 1;

direction = 'd';

} else if (mazeArray[currentY][currentX + 1] == 5) { //going right

currentX += 1;

direction = 'r';

}

break;

case 'd':

//know they were going down

if (mazeArray[currentY + 1][currentX] == 5) { //going down

currentY += 1;

direction = 'd';

} else if (mazeArray[currentY][currentX + 1] == 5) { //going right

currentX += 1;

direction = 'r';

} else if (mazeArray[currentY - 1][currentX] == 5) { //going up

currentY -= 1;

direction = 'u';

}

break;

case 'u':

//know they were going up

if (mazeArray[currentY][currentX + 1] == 5) { //going right

currentX += 1;

direction = 'r';

} else if (mazeArray[currentY - 1][currentX] == 5) { //going up

currentY -= 1;

direction = 'u';

} else if (mazeArray[currentY + 1][currentX] == 5) { //going down

currentY += 1;

direction = 'd';

}

break;

}

//situation for when cell is on side of board

} else if (currentX == sizeValue - 1 && currentY > 0 && currentY < sizeValue - 1) {

switch (direction) {

case 'r':

//know they were going right

if (mazeArray[currentY + 1][currentX] == 5) { //going down

currentY += 1;

direction = 'd';

} else if (mazeArray[currentY - 1][currentX] == 5) { //going up

currentY -= 1;

direction = 'u';

} else if (mazeArray[currentY][currentX - 1] == 5) { //going left

currentX -= 1;

direction = 'l';

}

break;

case 'l':

//know they were going left

if (mazeArray[currentY - 1][currentX] == 5) { //going up

currentY -= 1;

direction = 'u';

} else if (mazeArray[currentY][currentX - 1] == 5) { //going left

currentX -= 1;

direction = 'l';

} else if (mazeArray[currentY + 1][currentX] == 5) { //going down

currentY += 1;

direction = 'd';

}

break;

case 'd':

//know they were going down

if (mazeArray[currentY][currentX - 1] == 5) { //going left

currentX -= 1;

direction = 'l';

} else if (mazeArray[currentY + 1][currentX] == 5) { //going down

currentY += 1;

direction = 'd';

} else if (mazeArray[currentY - 1][currentX] == 5) { //going up

currentY -= 1;

direction = 'u';

}

break;

case 'u':

//know they were going up

if (mazeArray[currentY - 1][currentX] == 5) { //going up

currentY -= 1;

direction = 'u';

} else if (mazeArray[currentY][currentX - 1] == 5) { //going left

currentX -= 1;

direction = 'l';

} else if (mazeArray[currentY + 1][currentX] == 5) { //going down

currentY += 1;

direction = 'd';

}

break;

}

//situation for when cell is on side of board

} else if (currentY == 0 && currentX > 0 && currentX < sizeValue - 1) {

switch (direction) {

case 'r':

//know they were going right

if (mazeArray[currentY + 1][currentX] == 5) { //going down

currentY += 1;

direction = 'd';

} else if (mazeArray[currentY][currentX + 1] == 5) { //going right

currentX += 1;

direction = 'r';

} else if (mazeArray[currentY][currentX - 1] == 5) { //going left

currentX -= 1;

direction = 'l';

}

break;

case 'l':

//know they were going left

if (mazeArray[currentY][currentX - 1] == 5) { //going left

currentX -= 1;

direction = 'l';

} else if (mazeArray[currentY + 1][currentX] == 5) { //going down

currentY += 1;

direction = 'd';

} else if (mazeArray[currentY][currentX + 1] == 5) { //going right

currentX += 1;

direction = 'r';

}

break;

case 'd':

//know they were going down

if (mazeArray[currentY][currentX - 1] == 5) { //going left

currentX -= 1;

direction = 'l';

} else if (mazeArray[currentY + 1][currentX] == 5) { //going down

currentY += 1;

direction = 'd';

} else if (mazeArray[currentY][currentX + 1] == 5) { //going right

currentX += 1;

direction = 'r';

}

break;

case 'u':

//know they were going up

if (mazeArray[currentY][currentX + 1] == 5) { //going right

currentX += 1;

direction = 'r';

} else if (mazeArray[currentY][currentX - 1] == 5) { //going left

currentX -= 1;

direction = 'l';

} else if (mazeArray[currentY + 1][currentX] == 5) { //going down

currentY += 1;

direction = 'd';

}

break;

}

//situation for when cell is on side of board

} else if (currentY == sizeValue - 1 && currentX > 0 && currentX < sizeValue - 1) {

switch (direction) {

case 'r':

//know they were going right

if (mazeArray[currentY][currentX + 1] == 5) { //going right

currentX += 1;

direction = 'r';

} else if (mazeArray[currentY - 1][currentX] == 5) { //going up

currentY -= 1;

direction = 'u';

} else if (mazeArray[currentY][currentX - 1] == 5) { //going left

currentX -= 1;

direction = 'l';

}

break;

case 'l':

//know they were going left

if (mazeArray[currentY - 1][currentX] == 5) { //going up

currentY -= 1;

direction = 'u';

} else if (mazeArray[currentY][currentX - 1] == 5) { //going left

currentX -= 1;

direction = 'l';

} else if (mazeArray[currentY][currentX + 1] == 5) { //going right

currentX += 1;

direction = 'r';

}

break;

case 'd':

//know they were going down

if (mazeArray[currentY][currentX - 1] == 5) { //going left

currentX -= 1;

direction = 'l';

} else if (mazeArray[currentY][currentX + 1] == 5) { //going right

currentX += 1;

direction = 'r';

} else if (mazeArray[currentY - 1][currentX] == 5) { //going up

currentY -= 1;

direction = 'u';

}

break;

case 'u':

//know they were going up

if (mazeArray[currentY][currentX + 1] == 5) { //going right

currentX += 1;

direction = 'r';

} else if (mazeArray[currentY - 1][currentX] == 5) { //going up

currentY -= 1;

direction = 'u';

} else if (mazeArray[currentY][currentX - 1] == 5) { //going left

currentX -= 1;

direction = 'l';

}

break;

}

//situation for when cell is on corner of board

} else if (currentX == 0 && currentY == 0) {

switch (direction) {

case 'r':

//know they were going right

if (mazeArray[currentY + 1][currentX] == 5) { //going down

currentY += 1;

direction = 'd';

} else if (mazeArray[currentY][currentX + 1] == 5) { //going right

currentX += 1;

direction = 'r';

}

break;

case 'l':

//know they were going left

if (mazeArray[currentY + 1][currentX] == 5) { //going down

currentY += 1;

direction = 'd';

} else if (mazeArray[currentY][currentX + 1] == 5) { //going right

currentX += 1;

direction = 'r';

}

break;

case 'd':

//know they were going down

if (mazeArray[currentY + 1][currentX] == 5) { //going down

currentY += 1;

direction = 'd';

} else if (mazeArray[currentY][currentX + 1] == 5) { //going right

currentX += 1;

direction = 'r';

}

break;

case 'u':

//know they were going up

if (mazeArray[currentY][currentX + 1] == 5) { //going right

currentX += 1;

direction = 'r';

} else if (mazeArray[currentY + 1][currentX] == 5) { //going down

currentY += 1;

direction = 'd';

}

break;

}

//situation for when cell is on corner of board

} else if (currentX == 0 && currentY == sizeValue - 1) {

switch (direction) {

case 'r':

//know they were going right

if (mazeArray[currentY][currentX + 1] == 5) { //going right

currentX += 1;

direction = 'r';

} else if (mazeArray[currentY - 1][currentX] == 5) { //going up

currentY -= 1;

direction = 'u';

}

break;

case 'l':

//know they were going left

if (mazeArray[currentY - 1][currentX] == 5) { //going up

currentY -= 1;

direction = 'u';

} else if (mazeArray[currentY][currentX + 1] == 5) { //going right

currentX += 1;

direction = 'r';

}

break;

case 'd':

//know they were going down

if (mazeArray[currentY][currentX + 1] == 5) { //going right

currentX += 1;

direction = 'r';

} else if (mazeArray[currentY - 1][currentX] == 5) { //going up

currentY -= 1;

direction = 'u';

}

break;

case 'u':

//know they were going up

if (mazeArray[currentY][currentX + 1] == 5) { //going right

currentX += 1;

direction = 'r';

} else if (mazeArray[currentY - 1][currentX] == 5) { //going up

currentY -= 1;

direction = 'u';

}

break;

}

//situation for when cell is on corner of board

} else if (currentX == sizeValue - 1 && currentY == 0) {

switch (direction) {

case 'r':

//know they were going right

if (mazeArray[currentY + 1][currentX] == 5) { //going down

currentY += 1;

direction = 'd';

} else if (mazeArray[currentY][currentX - 1] == 5) { //going left

currentX -= 1;

direction = 'l';

}

break;

case 'l':

//know they were going left

if (mazeArray[currentY][currentX - 1] == 5) { //going left

currentX -= 1;

direction = 'l';

} else if (mazeArray[currentY + 1][currentX] == 5) { //going down

currentY += 1;

direction = 'd';

}

break;

case 'd':

//know they were going down

if (mazeArray[currentY][currentX - 1] == 5) { //going left

currentX -= 1;

direction = 'l';

} else if (mazeArray[currentY + 1][currentX] == 5) { //going down

currentY += 1;

direction = 'd';

}

break;

case 'u':

//know they were going up

if (mazeArray[currentY][currentX - 1] == 5) { //going left

currentX -= 1;

direction = 'l';

} else if (mazeArray[currentY + 1][currentX] == 5) { //going down

currentY += 1;

direction = 'd';

}

break;

}

//situation for when cell is on corner of board

} else if (currentX == sizeValue - 1 && currentY == sizeValue - 1) {

switch (direction) {

case 'r':

//know they were going right

if (mazeArray[currentY - 1][currentX] == 5) { //going up

currentY -= 1;

direction = 'u';

} else if (mazeArray[currentY][currentX - 1] == 5) { //going left

currentX -= 1;

direction = 'l';

}

break;

case 'l':

//know they were going left

if (mazeArray[currentY - 1][currentX] == 5) { //going up

currentY -= 1;

direction = 'u';

} else if (mazeArray[currentY][currentX - 1] == 5) { //going left

currentX -= 1;

direction = 'l';

}

break;

case 'd':

//know they were going down

if (mazeArray[currentY][currentX - 1] == 5) { //going left

currentX -= 1;

direction = 'l';

} else if (mazeArray[currentY - 1][currentX] == 5) { //going up

currentY -= 1;

direction = 'u';

}

break;

case 'u':

//know they were going up

if (mazeArray[currentY - 1][currentX] == 5) { //going up

currentY -= 1;

direction = 'u';

} else if (mazeArray[currentY][currentX - 1] == 5) { //going left

currentX -= 1;

direction = 'l';

}

break;

}

}

//setting cell to shortest path which will display as green

mazeArray[currentY][currentX] = 4;

//increasing position counter by one

positionCounter += 1;

//setting position of green cell

positionArray[currentY][currentX] = positionCounter;

//recursively sending to method MakePath

MakePath();

}

}

//declaring class used for the grid gui

public class GridPane extends JPanel {

//declaring public used for mouse events

public GridPane() {

//declaring an array list used for grid gui

cells = new ArrayList<>(sizeValue \* sizeValue);

//creating a mouse listener

addMouseListener(new MouseAdapter() {

@Override

//execute when mouse is clicked

public void mouseClicked(MouseEvent e) {

//do only if colour mode is 0, or 1

if (colourMode == 0 || colourMode == 1) {

//declaring and setting x and y variables

int horizontalClickPosition = (e.getX() - xOffset) / (getWidth() / sizeValue);

int verticalClickPosition = (e.getY() - yOffset) / (getHeight() / sizeValue);

//do only if cell in area is a path tile

if (verticalClickPosition >= 0 && verticalClickPosition <= sizeValue - 1 && horizontalClickPosition >= 0 && horizontalClickPosition <= sizeValue - 1) {

if (mazeArray[verticalClickPosition][horizontalClickPosition] == 0) {

//only execute following lines of code if criteria are met

if (colourMode == 0) {

//setting the start y coordinate

startY = currentY = verticalClickPosition;

//setting the start x coordinate

startX = currentX = horizontalClickPosition;

//setting starting direction in maze

if (currentX > 0 && currentX < sizeValue - 1 && currentY > 0 && currentY < sizeValue - 1) {

if (mazeArray[currentY][currentX + 1] == 0) {

direction = 'r';

} else if (mazeArray[currentY + 1][currentX] == 0) {

direction = 'd';

} else if (mazeArray[currentY][currentX - 1] == 0) {

direction = 'l';

} else if (mazeArray[currentY - 1][currentX] == 0) {

direction = 'u';

}

} else if (currentX == 0 && currentY > 0 && currentY < sizeValue - 1) {

if (mazeArray[currentY][currentX + 1] == 0) {

direction = 'r';

} else if (mazeArray[currentY + 1][currentX] == 0) {

direction = 'd';

} else if (mazeArray[currentY - 1][currentX] == 0) {

direction = 'u';

}

} else if (currentX == sizeValue - 1 && currentY > 0 && currentY < sizeValue - 1) {

if (mazeArray[currentY + 1][currentX] == 0) {

direction = 'd';

} else if (mazeArray[currentY][currentX - 1] == 0) {

direction = 'l';

} else if (mazeArray[currentY - 1][currentX] == 0) {

direction = 'u';

}

} else if (currentX > 0 && currentX < sizeValue - 1 && currentY == 0) {

if (mazeArray[currentY][currentX + 1] == 0) {

direction = 'r';

} else if (mazeArray[currentY + 1][currentX] == 0) {

direction = 'd';

} else if (mazeArray[currentY][currentX - 1] == 0) {

direction = 'l';

}

} else if (currentX > 0 && currentX < sizeValue - 1 && currentY == sizeValue - 1) {

if (mazeArray[currentY][currentX + 1] == 0) {

direction = 'r';

} else if (mazeArray[currentY][currentX - 1] == 0) {

direction = 'l';

} else if (mazeArray[currentY - 1][currentX] == 0) {

direction = 'u';

}

} else if (currentX == 0 && currentY == 0) {

if (mazeArray[currentY][currentX + 1] == 0) {

direction = 'r';

} else if (mazeArray[currentY + 1][currentX] == 0) {

direction = 'd';

}

} else if (currentX == 0 && currentY == sizeValue - 1) {

if (mazeArray[currentY][currentX + 1] == 0) {

direction = 'r';

} else if (mazeArray[currentY - 1][currentX] == 0) {

direction = 'u';

}

} else if (currentX == sizeValue - 1 && currentY == 0) {

if (mazeArray[currentY + 1][currentX] == 0) {

direction = 'd';

} else if (mazeArray[currentY][currentX - 1] == 0) {

direction = 'l';

}

} else if (currentX == sizeValue - 1 && currentY == sizeValue - 1) {

if (mazeArray[currentY][currentX - 1] == 0) {

direction = 'l';

} else if (mazeArray[currentY - 1][currentX] == 0) {

direction = 'u';

}

}

//setting start tile

mazeArray[verticalClickPosition][horizontalClickPosition] = 2;

//increasing colourMode by one

colourMode += 1;

} else if (colourMode == 1) {

//setting the end y coordinate

endY = verticalClickPosition;

//setting the end x coordinate

endX = horizontalClickPosition;

//setting end tile

mazeArray[verticalClickPosition][horizontalClickPosition] = 3;

//increasing colourMode by one

colourMode += 1;

//sending to method StartSolver

StartSolver();

}

}

}

}

}

});

//creating new mouse handler

MouseAdapter mouseHandler;

mouseHandler = new MouseAdapter() {

//if user moves mouse execute following line of code in order to show temporary colour where a tile would be if user mouse clicked

@Override

public void mouseMoved(MouseEvent e) {

int width = getWidth();

int height = getHeight();

int cellWidth = width / sizeValue;

int cellHeight = height / sizeValue;

selectedCell = null;

if (e.getX() >= xOffset && e.getY() >= yOffset) {

int column = (e.getX() - xOffset) / cellWidth;

int row = (e.getY() - yOffset) / cellHeight;

if (column >= 0 && row >= 0 && column < sizeValue && row < sizeValue) {

selectedCell = new Point(column, row);

}

}

repaint();

}

};

addMouseMotionListener(mouseHandler);

}

//setting size of the grid gui

@Override

public Dimension getPreferredSize() {

return new Dimension(guiDisplay, guiDisplay + 49);

}

//protected void used for setting cell colour

@Override

protected void paintComponent(Graphics g) {

//following lines used to determine x and y coordinates

super.paintComponent(g);

Graphics2D g2d = (Graphics2D) g.create();

int width = getWidth();

int height = getHeight();

int cellWidth = width / sizeValue;

int cellHeight = height / sizeValue;

xOffset = (width - (sizeValue \* cellWidth)) / 2;

yOffset = (height - (sizeValue \* cellHeight)) / 2;

if (cells.isEmpty()) {

for (int row = 0; row < sizeValue; row++) {

for (int col = 0; col < sizeValue; col++) {

Rectangle cell = new Rectangle(

xOffset + (col \* cellWidth),

yOffset + (row \* cellHeight),

cellWidth,

cellHeight);

cells.add(cell);

}

}

}

//used for showing temporary cell colour where cursor is hovering and when if clicked would become permanent colour

if (selectedCell != null && (colourMode == 0 || colourMode == 1)) {

if (selectedCell.x + (selectedCell.y \* sizeValue) <= sizeValue \* sizeValue) {

int index = selectedCell.x + (selectedCell.y \* sizeValue);

Rectangle cell = cells.get(index);

if (colourMode == 0) {

g2d.setColor(Color.RED);

} else if (colourMode == 1) {

g2d.setColor(Color.BLUE);

}

g2d.fill(cell);

}

}

//drawing grey outlines of the cells

g2d.setColor(Color.GRAY);

cells.stream().forEach((cell) -> {

g2d.draw(cell);

});

//creating temporary time variable

int tempTime = time;

//setting value of time

time = timingSlider.getValue();

//sending to method Save only if the time is different

if (tempTime != time) {

Save();

}

boolean cancel = false;

//2d array used for setting colour of cell

for (int vertical = 0; vertical < sizeValue; vertical++) {

for (int horizontal = 0; horizontal < sizeValue; horizontal++) {

Rectangle cell = cells.get(horizontal + vertical \* sizeValue);

switch (mazeArray[vertical][horizontal]) {

case 1:

//if cell is of type 1 set colour to black

g2d.setColor(Color.BLACK);

g2d.fill(cell);

break;

case 2:

//if cell is of type 2 set colour to red

g2d.setColor(Color.RED);

g2d.fill(cell);

break;

case 3:

//if cell is of type 3 set colour to blue

g2d.setColor(Color.BLUE);

g2d.fill(cell);

break;

case 4:

//if cell is of type 4 set colour to green

g2d.setColor(Color.GREEN);

//checking if all the green cells are currently filled in or not

if (positionArray[vertical][horizontal] == positionCounter && cancel == false) {

//filling in current and trailing green cells

for (int vertical2 = 0; vertical2 < sizeValue; vertical2++) {

for (int horizontal2 = 0; horizontal2 < sizeValue; horizontal2++) {

if (positionArray[vertical2][horizontal2] >= positionCounter) {

g2d.fill(cells.get(horizontal2 + vertical2 \* sizeValue));

}

}

}

//sleeping for specified amount of time

try {

Thread.sleep(time);

} catch (InterruptedException ex) {

Logger.getLogger(Maze.class.getName()).log(Level.SEVERE, null, ex);

}

//reducing position counter by one

positionCounter -= 1;

//setting cancel to true

cancel = true;

} else if (positionCounter == 0) {

//filling in green cells

g2d.fill(cell);

}

break;

}

repaint();

}

}

}

}

//declaring private void method used for loading from file io

private void Load() {

try {

//trying to create file

Files.createFile(file);

//executed if file already exists

} catch (FileAlreadyExistsException x) {

//file is read from and saved to variable saveFile is file already exists

try (InputStream in = Files.newInputStream(file);

BufferedReader reader = new BufferedReader(new InputStreamReader(in))) {

String line;

while ((line = reader.readLine()) != null) {

//content of file is saved to saveFile

saveFile = line;

}

} catch (IOException y) {

System.err.println(y);

}

} catch (IOException x) {

System.err.println(x);

}

//if the file does not contain anything since it was just created, default variables are used for save file

if (saveFile == null) {

saveFile = "10 100 0 0 0 0 0 1 0 0 0 0 1 1 0 1 0 1 0 1 1 0 1 1 0 1 0 1 0 1 1 0 1 0 1 0 0 0 1 1 0 0 0 0 0 1 1 0 0 1 1 0 0 1 0 1 1 0 1 1 0 0 1 0 0 0 0 0 0 0 1 0 1 1 1 0 1 0 1 0 0 0 1 0 0 0 1 1 1 1 1 0 0 0 1 0 0 0 1 0 0 0";

}

//a String array is created and each part of the array is saved to from saveFile seperated by spaces

split = saveFile.split("\\s+");

//variable size is the first number

sizeValue = parseInt(split[0], 10);

//variable time is the second number

time = parseInt(split[1], 10);

}

//declaring private void method used for saving with file io

private void Save() {

//saveFile is created using the main variables seperated by spaces

saveFile = sizeValue + " " + time;

for (int vertical = 0; vertical < sizeValue; vertical++) {

for (int horizontal = 0; horizontal < sizeValue; horizontal++) {

saveFile += " ";

if (mazeArray[vertical][horizontal] == 0 || mazeArray[vertical][horizontal] == 1) {

saveFile += mazeArray[vertical][horizontal];

} else {

saveFile += 0;

}

}

}

//saveFile is converted to byte data

byte data[] = saveFile.getBytes();

//byte data is saved to file using file io

try (OutputStream out = new BufferedOutputStream(

Files.newOutputStream(file, WRITE, TRUNCATE\_EXISTING))) {

out.write(data, 0, data.length);

} catch (IOException x) {

System.err.println(x);

}

}

}